Algorithm Analysis using Big-Oh Notation

Big-Oh Definition

Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers. We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq c \cdot g(n), \text{ for } n \geq n_0.$$

Remarks

1. We use the Big-Oh notation to analyse the running time of algorithms. This analysis is also referred to as worst-case analysis (because we always consider the worst-case running time when performing the analysis), and order of growth analysis (because the function identified in this analysis characterises the rate of growth of the running time of the algorithm, with respect to the input size).

2. To analyse the running time of any algorithm, we count the number of primitive operations of that algorithm in the worst case execution of the algorithm with respect to the size of the input. The most significant term of the resultant function allows us to characterise the worst-case running time of the algorithm using the Big-Oh notation.

3. Note that unless explicitly asked, it is not necessary to perform a precise count of the number of primitive operations. Instead, we can directly try and identify the most significant term and use it to characterise the algorithm's running time. This approach is illustrated in the examples that follow.

Examples

1. Justify that $20n^3 + 10n\log n + 5$ is $O(n^3)$.

Soln.

$20n^3 + 10n\log n + 5 \leq 20n^3 + 10n^3 + 5n^3$, for $n \geq 1$
(as $20n^3 = 20n^3$; $10n\log n \leq 10n^3$, for $n \geq 1$; $5 \leq 5n^3$, for $n \geq 1$)
Hence, $20n^3 + 10n\log n + 5 \leq 35n^3$, for $n \geq 1$
Hence, $20n^3 + 10n\log n + 5$ is $O(n^3)$, for $c = 35$ and $n_0 = 1$.
Hence, $20n^3 + 10n\log n$ is $O(n^3)$.

Remarks: Given any equation, to represent it in the Big-Oh notation, we need to use the Big-Oh definition, and identify appropriate values of c and n_0 to satisfy the conditions of the definition.

2. Suppose $f(x) = x^2 + 2x + 2$ and $g(x) = x^2$. Prove that $f(x) = O(g(x))$ and $g(x)$ is $O(f(x))$.

Soln.

$f(x) = x^2 + 2x + 2 \leq 3x^2$, for $x \geq 2$
Thus, for $c = 3$ and $x_0 = 2$, we have $f(x) = x^2 + 2x + 2 = O(g(x))$.
Also, $g(x) = x^2 < x^2 + 2x + 2$, for $x \geq 0$
Thus, for $c = 1$ and $x_0 = 0$, we have $g(x) = x^2 = O(f(x))$.

3. Use the Big-Oh notation to characterise the order of growth of the following algorithm.

```
Algorithm 1
Input: Array arr, array size n, array index i, value x
Output: true, if arr[i] = x, otherwise, false
1. if i < n
2.      if arr[i] == x
3.              return true
4. return false
```

Soln.

This algorithm is simply checking whether a given value is equal to the value stored at the given index of an array. Regardless of how large the size of the array is, since accessing the value of an array at any index is a constant time operation, the order of growth of this algorithm is $O(1)$ (i.e., this is a constant time algorithm).

4. Characterise the order of growth of the following Java method using Big-Oh notation.

```
public void print3(int n)
{
        for(int i = 0; i < 3; i++)
                System.out.println(n);
}
```

Soln.

The above Java method takes an integer input and prints that input 3 times. We note that the loop runs for a fixed number of times (3), and does not depend upon the parameter size. Since printing a number on the console is a primitive operation (i.e., it takes a fixed amount of time), the order of growth of the above Java method is $O(1)$.

Remarks: As shown in the example above, just because a method or algorithm has a loop does not mean that the order of growth is linear (or higher). In this example, the loop is running a fixed number of times. Thus, as long as the statements inside the loop are all primitive operations, overall, the entire loop can be estimated to run in constant time.

5. Perform a worst-case analysis of the following algorithm.

```
Algorithm 2
Input: Sorted array A, and target value x
Output: Index of x in A, or -1 if not found
1. low = 0
2. high = len(A) – 1
3. while low <= high do
4.      mid = (low+high) / 2
5.      if A[mid] < x then
6.              low = mid + 1
7.      else if A[mid] > x then
8.              high = mid – 1
9.      else
10.             return mid
11. return -1
```

Soln.

This algorithm represents the iterative implementation of binary search. Note that inside the loop if we do not find the value x at A[mid], we are essentially halving the size of the array. Thus, we have, 2 * 2 * 2 *....* 2 = 2^x <= n, i.e., $\log_2(2^x) = x <= \log_2(n)$ (where x represents the number of times n can be halved, i.e., the number of iterations of the while loop). Thus, the above algorithm is $O(\log_2(n))$ (where n is the size of the array).

6. Calculate the order of growth of the following algorithm.

```
Algorithm 3
Input: Array A, and target value x
Output: Index of x in A, or -1 if not found
1. for i from 0 to len(A) do
2.         if A[i] = x then
3.                 return i
4. return -1
```

Soln.

Clearly this algorithm is performing linear search, i.e, it is going through each element in the given array and comparing it with the target value (x). In case a match occurs, the index at which the match occurs is returned. In case the match does not occur, the value -1 is returned.

In analysing this algorithm, we note that the worst case performance will be observed when the target value (x) is not found in the array. In such a scenario, the algorithm will compare each value in the array to the target value unsuccessfully. In this worst-case scenario, the comparison in line 2 of the algorithm will be performed a maximum of n times, where n is the size of the array. Thus, the order of growth of this algorithm is $O(n)$.

7. What is the time complexity of the following Java method?

```
public void method(int n)
{
        for(int i = 1; i <= n; i++)
                for(int j = 1; j <= n; j++)
                        System.out.println(i + " " + j);
}
```

Soln.

The method shown above is essentially printing all the permutations of two numbers having values from 1 to n. It should be clear that the print statement inside the nested for loops will be executed n^2 times. Thus, this method has a worst-case time complexity of $O(n^2)$.

8. Analyse the following Java method using Big-Oh notation.

```
public void unique(int array[])
{
        for(int i = 0; i < array.length - 1; i++)
                for(int j = i+1; j < array.length; j++)
                        if(array[i] == array[j])
                                return false;
        return true;
}
```

Soln.

The method shown above is given an array and returns true or false depending upon whether the array elements are all unique or not.

In analysing the algorithm, the first thing to note is that the running time of the algorithm depends upon the size of the array provided as input. Additionally, we can immediately identify that the worst case occurs when the maximum possible comparisons are made, i.e, when each element in the provided array is unique (and thus, the method returns false).

Having identified the worst case, we simply need to calculate the number of times the comparison inside the inner loop (i.e., the if statement comparison) is being made. Assuming array.length to be n, we observe that the outer loop runs from 0 to n-2 , and the inner loop runs from i+1 to n-1 for each value of i. To calculate the total number of comparisons made (in the worst-case), we sum the number of iterations of the inner loop to get

$$(n-1) + (n-2) + ... + 2 + 1 = n(n-1)/2$$

Using the Big-Oh definition, we have $f(n) = n(n-1)/2$ is $O(n^2)$. Thus the above Java method has a worst-case time complexity of $O(n^2)$.